Chapitre 7

OSA – RTOS Fonctionnement

7.1 **Introduction aux systèmes d'exploitation temps réel**

Ce chapitre constitue une étape charnière dans la compréhension des systèmes embarqués complexes, en opérant une transition fondamentale entre une approche de programmation séquentielle, souvent basée sur une boucle principale (super-loop), et une architecture logicielle multitâche orchestrée par un noyau temps réel. Jusqu'à présent, les applications développées reposaient sur une logique où le microcontrôleur exécutait les instructions les unes après les autres, avec des interruptions venant ponctuellement perturber ce flux linéaire. Cependant, à mesure que les applications gagnent en complexité – intégrant simultanément acquisition de capteurs, communication, interface utilisateur et contrôle d'actionneurs – cette approche atteint rapidement ses limites en termes de réactivité, de maintenabilité et de respect de contraintes temporelles. C'est ici qu'intervient le système d'exploitation temps réel (RTOS – Real-Time Operating System), un logiciel intermédiaire qui agit comme un chef d'orchestre, permettant de découper l'application en plusieurs blocs fonctionnels indépendants appelés tâches, et de gérer leur exécution de manière à donner l'illusion d'un parallélisme tout en respectant des échéances temporelles strictes. Nous prendrons comme support concret un RTOS léger particulièrement adapté aux microcontrôleurs aux ressources limitées, tel que OSA (qui est un RTOS simple et pédagogique) ou FreeRTOS (le standard industriel de facto), afin d'illustrer de manière pratique et progressive les concepts fondamentaux qui seront développés tout au long de ce chapitre.

7.2 **La notion de tâche : unité fondamentale du parallélisme logiciel**

Au cœur de tout système RTOS se trouve la notion de tâche (ou thread), qui représente l'unité de base d'exécution. Une tâche peut être conceptualisée comme un petit programme indépendant, doté de sa propre séquence d'instructions, de son propre contexte d'exécution, et de sa propre pile mémoire (stack). Contrairement à une approche où une seule fonction main() contrôle l'ensemble de l'application, la conception basée sur les tâches consiste à identifier les activités concurrentes du système – par exemple, une tâche pour lire un capteur de température, une autre pour gérer un écran d'affichage, une troisième pour surveiller un bouton poussoir, et une quatrième pour contrôler un moteur – et à les implémenter chacune comme une fonction autonome s'exécutant dans sa propre boucle infinie. Lors de la création d'une tâche, le développeur doit spécifier plusieurs paramètres essentiels : le nom de la fonction qui constitue le corps de la tâche, sa priorité relative par rapport aux autres tâches, la taille de la pile mémoire qui lui est allouée (un paramètre critique car une taille insuffisante conduit à un débordement, tandis qu'une taille excessive gaspille la précieuse RAM du microcontrôleur), et éventuellement des paramètres d'initialisation. Dans notre RTOS d'exemple, la création d'une tâche s'effectue généralement via une fonction du type OS_TaskCreate() ou xTaskCreate(), qui prépare le noyau à prendre en charge cette nouvelle entité. Il est important de souligner que chaque tâche possède son propre contexte,

c'est-à-dire l'ensemble des valeurs des registres du processeur, du compteur programme et de la pile à un instant donné ; c'est la capacité du RTOS à sauvegarder et restaurer ces contextes lors des changements de tâche qui rend possible l'illusion du parallélisme.

7.3 Les états d'une tâche : le cycle de vie

Une tâche ne se trouve pas constamment en train d'utiliser le processeur ; elle évolue au cours du temps à travers différents états qui définissent son cycle de vie, et c'est précisément la gestion de ces états par l'ordonnanceur qui confère au système sa flexibilité et son efficacité. Le premier état est l'état prêt (ready) : une tâche dans cet état dispose de toutes les ressources nécessaires pour s'exécuter (notamment sa pile et son contexte sont intacts) et attend seulement que le processeur lui soit alloué. L'état en cours (running) est celui de la tâche qui possède effectivement le contrôle du processeur à un instant donné ; sur un microcontrôleur monocœur, il ne peut y avoir qu'une seule tâche dans cet état à la fois. L'état bloqué (blocked) survient lorsqu'une tâche attend un événement externe ou interne avant de pouvoir poursuivre son exécution – par exemple, l'attente d'une temporisation expirant, d'une donnée arrivant sur une file de messages, ou d'un sémaphore devenant disponible. Une tâche bloquée n'est pas éligible pour l'exécution tant que l'événement attendu ne s'est pas produit, ce qui est extrêmement efficace car cela évite des attentes actives qui consommeraient inutilement du temps processeur. Enfin, l'état suspendu (suspended) est un état particulier où une tâche est volontairement mise en sommeil par une autre tâche ou par elle-même, généralement à des fins de débogage ou de gestion dynamique des ressources ; contrairement à l'état bloqué, la tâche suspendue ne sera pas automatiquement réveillée par un événement, mais nécessitera une action explicite de reprise. La transition entre ces états est gérée par des fonctions spécifiques du RTOS : OS_TaskDelay() ou vTaskDelay() pour bloquer une tâche pendant une durée déterminée, OS_SemaphoreWait() pour attendre un sémaphore, ou OS_TaskSuspend() et OS_TaskResume() pour les opérations de suspension et de reprise.

7.4 L'ordonnanceur : le cœur décisionnel du RTOS

L'ordonnanceur (scheduler) constitue le cœur du système d'exploitation temps réel ; il s'agit du composant logiciel responsable de décider, à tout moment, quelle tâche parmi toutes celles qui sont prêtes doit être exécutée par le processeur. Cette décision n'est pas arbitraire mais repose sur un algorithme d'ordonnancement, et c'est la nature de cet algorithme qui définit les caractéristiques temps réel du système. Dans la grande majorité des RTOS utilisés dans l'industrie (FreeRTOS, OSA, embOS, etc.), l'algorithme retenu est celui de l'ordonnancement préemptif basé sur les priorités. Le principe est simple mais puissant : chaque tâche se voit attribuer une priorité numérique (par exemple, une valeur entière allant de 0 pour la priorité la plus basse à un nombre défini pour la priorité la plus haute), et l'ordonnanceur s'assure en permanence que la tâche prête ayant la plus haute priorité est toujours celle qui est en cours d'exécution. La nature "préemptive" de cet algorithme signifie que si une tâche de priorité plus élevée devient prête (par exemple, parce qu'elle était bloquée et que l'événement attendu vient de se produire), l'ordonnanceur a le droit d'interrompre immédiatement la tâche en cours d'exécution, même si celle-ci n'a pas terminé son travail, pour lui substituer la tâche plus prioritaire. Cette préemption est un mécanisme fondamental pour garantir la réactivité des systèmes temps réel : elle assure que les événements critiques sont traités avec la latence

minimale, sans avoir à attendre que la tâche courante se termine volontairement. Il est important de noter que si plusieurs tâches partagent la même priorité, l'ordonnanceur peut adopter différentes stratégies selon la configuration : l'ordonnancement circulaire (round-robin) où chaque tâche s'exécute à tour de rôle pendant un quantum de temps fixe, ou l'ordonnancement premier-arrivé-premier-servi (FIFO). Dans notre RTOS d'exemple, nous verrons comment configurer les priorités lors de la création des tâches, et comment observer le comportement préemptif à travers des exemples concrets mettant en œuvre des tâches de différentes priorités interagissant entre elles.

7.5 Implémentation pratique et mécanismes internes

Pour consolider ces concepts théoriques, ce chapitre proposera une mise en œuvre pratique progressive permettant d'observer concrètement le fonctionnement du RTOS sur le microcontrôleur. Nous commencerons par un exemple minimaliste consistant à créer deux tâches simples, chacune réalisant une action élémentaire (par exemple, faire clignoter une LED à des fréquences différentes) et utilisant un blocage par temporisation (OS_TaskDelay) pour libérer le processeur entre deux actions. Cet exemple permettra de visualiser comment les tâches s'exécutent alternativement et comment l'ordonnanceur répartit le temps processeur. Nous introduirons ensuite une troisième tâche avec une priorité supérieure, simulant un événement critique (comme la détection d'une urgence via un bouton poussoir) afin de démontrer le mécanisme de préemption : même si la tâche de plus basse priorité est en train de s'exécuter, l'apparition de l'événement conduira à l'interrompre immédiatement au profit de la tâche prioritaire. Nous analyserons également en détail les mécanismes internes qui rendent cette préemption possible : le "tick" du RTOS, qui est une interruption périodique (généralement générée par un timer) servant d'horloge interne au noyau, et qui déclenche à chaque expiration l'évaluation des priorités et éventuellement le changement de contexte. Le changement de contexte, ou "context switch", est l'opération critique par laquelle le RTOS sauvegarde l'état complet (registres CPU, compteur programme, pointeur de pile) de la tâche interrompue, restaure celui de la nouvelle tâche élue, et redonne la main à celle-ci. Nous verrons comment ce mécanisme, bien que complexe, est encapsulé par le RTOS et totalement transparent pour le développeur applicatif, qui n'a qu'à se concentrer sur la logique métier de chaque tâche sans avoir à gérer manuellement la bascule entre elles. À l'issue de ce chapitre, l'apprenant sera capable de structurer une application embarquée sous forme de tâches indépendantes, de configurer leurs priorités, et de comprendre le comportement temporel résultant de l'ordonnancement préemptif, posant ainsi les bases solides nécessaires pour aborder le chapitre suivant dédié aux services avancés du RTOS comme la synchronisation et la communication inter-tâches.